

Advanced validation

lesson6

So far we have only done some basic validations using the `@Required` annotation. Sometimes it's convenient to write our own logic for the validation.

In appendix C we introduce the `@Required` annotation as a basic way to implement validation logic. In this lesson we are going to describe custom validation methods which allow us to add specific business logic to your application.

6.1 Validation alternatives

We are going to enhance your code with this logic: if the orders are not delivered yet, then the user cannot assign them to an invoice. That is, only delivered orders can be associated with an invoice.

6.1.1 Adding delivered property to Order

First you have to add a new property to the `Order` entity. The delivered property (listing 6.1).

Listing 6.1 New delivered property in Order entity

```
private boolean delivered;

public boolean isDelivered() {
    return delivered;
}

public void setDelivered(boolean delivered) {
    this.delivered = delivered;
}
```

Update the database schema now. Then execute the SQL statement in listing 6.2 against the database. You can use the Database perspective in Eclipse to do so (see lesson 1).

Listing 6.2 Set the delivered column of CommercialDocument table to false

```
update CommercialDocument
set delivered = false
```

Moreover it's necessary to add the delivered property to the view. Modify the `Order` view as in listing 6.3.

Listing 6.3 Order view modified to include the new delivered property

```
@Views({
    @View( extendsView="super.DEFAULT",
        members="delivered; invoice { invoice }" // delivered added
    ),
    ...
})
public class Order extends CommercialDocument {
```

There is a new delivered property now which indicates the delivery state of an order. Try the new code and mark some of the existing orders as delivered.

6.1.2 Validating with @EntityValidator

Up to now the user can add any order to any invoice from the Invoice module, and he can associate a particular invoice with any order from the Order module. We are going to restrict this: only delivered orders are allowed to be added to an invoice.

The first alternative to implement this validation is by using an @EntityValidator. This annotation allows you to assign the desired validation logic to your entity. Let's annotate the Order entity as in listing 6.4.

Listing 6.4 @EntityValidator for Order entity

```
@EntityValidator(
    value=DeliveredToBeInInvoiceValidator.class, // The class with the validation logic
    properties= {
        @PropertyValue(name="year"),           // The content of these properties
        @PropertyValue(name="number"),         // is moved from the Order entity
        @PropertyValue(name="invoice"),         // to the validator before
        @PropertyValue(name="delivered")        // executing the validation
    }
)
public class Order extends CommercialDocument {
```

Every time an Order object is created or modified an object of type DeliveredToBeInInvoiceValidator is created. Then its properties year, number, invoice and delivered are initialized with the properties of the same name from the Order object. After that, the validate() method of the validator is executed. You can see the code for the validator in listing 6.5.

Listing 6.5 Validator to validate that an order must be delivered to be in an invoice

```
package org.openxava.invoicing.validators; // In 'validators' package

import org.openxava.invoicing.model.*;
import org.openxava.util.*;
import org.openxava.validators.*;

public class DeliveredToBeInInvoiceValidator
    implements IValidator { // Must implement IValidator

    private int year; // Properties to be injected from Order
    private int number;
    private boolean delivered;
    private Invoice invoice;

    public void validate(Messages errors) // The validation logic
        throws Exception
    {
        if (invoice == null) return;
    }
```

```

    if (!delivered) {
        errors.add( // By adding messages to errors the validation will fail
            "order_must_be_delivered", // An id from i18n file
            year, number); // Arguments for the message
    }
}

// Getters and setters for year, number, delivered and invoice
...
}

```

The validation logic is absolutely straightforward: if an invoice is present and this order is not marked as delivered we add an error message, so the validation will fail. You should add the error message in the *Invoicing/i18n/Invoicing-messages_en.properties* file. Just as in listing 6.6.

Listing 6.6 Internationalization for error in Invoicing-messages_en.properties

```

# Messages for the Invoicing application
order_must_be_delivered=Order {0}/{1} must be delivered in order to be added to
an Invoice

```

Now you can try to add orders to an invoice with the application, you will see how the undelivered orders are rejected, as shown in figure 6.1.

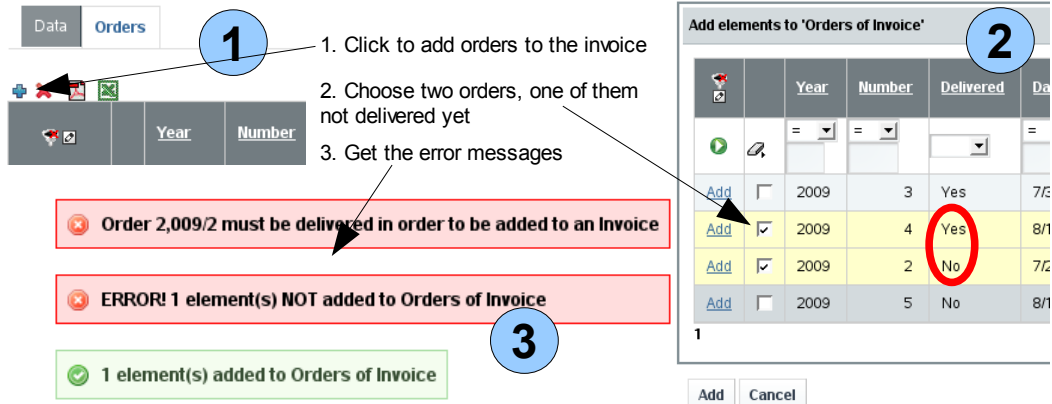


Figure 6.1 Adding not delivered orders produces validation errors

Your validation is implemented correctly with `@EntityValidator`. It's not difficult, but a little “verbose”, because you need to write a “fully featured” new class merely to add 2 lines of code logic. Let's learn other ways to do the same validation.

6.1.3 Validating with a JPA callback method

We're going to try another, maybe even simpler, way to do this validation:

we'll transfer the validation logic from the validator class into the Order entity itself, in this case in a `@PreUpdate` method.

First, remove the `DeliveredToBeInInvoiceValidator` class from your project. Then remove the `@EntityValidator` annotation from your Order entity (listing 6.7) as well.

Listing 6.7 Removing the `@EntityValidator` annotation from the Order entity

```
@EntityValidator(value=DeliveredToBeInInvoiceValidator.class,
    properties= { // Remove the @EntityValidator annotation
        @PropertyValue(name="year"),
        @PropertyValue(name="number"),
        @PropertyValue(name="invoice"),
        @PropertyValue(name="delivered")
    }
}

public class Order extends CommercialDocument {
```

After that we're going to add the validation again, but now inside the Order class itself. Add the `validate()` method in listing 6.8 to your Order class.

Listing 6.8 JPA callback method for validating in the Order entity

```
@PreUpdate // Just before updating the database
private void validate() throws Exception {
    if (invoice != null && !isDelivered()) { // The validation logic
        throw new InvalidStateException( // The validation exception from
            new InvalidValue[] { // Hibernate Validator framework
                new InvalidValue(
                    "Order must be delivered",
                    getClass(), "delivered",
                    true, this)
            }
        );
    }
}
```

Before saving an order this validation will be executed. If it fails an `InvalidStateException` is thrown by the Hibernate Validator framework. This way OpenXava will know that the exception is a validation exception. The cumbersome part of this solution is that `InvalidStateException` requires an array of `InvalidValue` objects. The good part is that with only one additional method inside the entity you have implemented the validation.

6.1.4 Validating in the setter

Another alternative to do your validation is to put the validation logic inside the setter method. That's a simple approach. First, remove the `validate()` method from the Order entity, and modify the `setInvoice()` method in the way you see in listing 6.9.

Listing 6.9 Validation inside the setter method for invoice in Order

```

public void setInvoice(Invoice invoice) {
    if (invoice != null && !isDelivered()) { // The validation logic
        throw new InvalidStateException( // The validation exception from
            new InvalidValue[] { // Hibernate Validator framework
                new InvalidValue(
                    "Order must be delivered",
                    getClass(), "delivered",
                    true, this)
            }
        );
    }
    this.invoice = invoice; // The regular setter assignment
}

```

This works exactly the same way as the two other options. This is like the `@PrePersist` alternative, only that it does not depend on JPA, it's a basic Java implementation.

6.1.5 Validating with Hibernate Validator

As a last option we are going to do the shortest one: The validation logic is put into a boolean method annotated with the `@AssertTrue` Hibernate Validator annotation.

To implement this alternative first remove the validation logic from the `setInvoice()` method. Then add the `isDeliveredToBeInInvoice()` method of listing 6.10 to your `Order` entity.

Listing 6.10 Validating Order using a `@AssertTrue` annotation

```

@AssertTrue // Before saving it asserts if this method returns true, if not it throws an exception
private boolean isDeliveredToBeInInvoice() {
    return invoice == null || isDelivered(); // The validation logic
}

```

This is the simplest way to validate, because the method with the validation only has to be annotated. The Hibernate Validator is responsible for calling this method when saving takes place, and throws the corresponding `InvalidStateException` if the validation does not succeed.

6.1.6 Validating on removal with `@RemoveValidator`

The validations we have seen until now are processed when the entity is modified, but sometimes it's useful or it's required to process the validation before the removal of the entity, and to use the validation to cancel the entity removal.

We are going to modify the application to reject the removal of an order if it has an invoice associated. To achieve this annotate your `Order` entity with `@RemoveValidator`, just as shown in listing 6.11.

Listing 6.11 @RemoveValidator for Order entity

```
@RemoveValidator(OrderRemoveValidator.class) // The class with the validation
public class Order extends CommercialDocument {
```

Now, before removing an order the logic in `OrderRemoveValidator` is executed, and if validation fails the order is not removed. Let's look at the code for the validator in listing 6.12.

Listing 6.12 Validator to validate if an order can be removed

```
package org.openxava.invoicing.validators; // In 'validators' package

import org.openxava.invoicing.model.*;
import org.openxava.util.*;
import org.openxava.validators.*;

public class OrderRemoveValidator
    implements IRemoveValidator { // Must implement IRemoveValidator

    private Order order;

    public void setEntity(Object entity) // The entity to remove will be injected
        throws Exception                // with this method before validating
    {
        this.order = (Order) entity;
    }

    public void validate(Messages errors) // The validation logic
        throws Exception
    {
        if (order.getInvoice() != null) {
            errors.add("cannot_delete_order_with_invoice"); // By adding messages
                                                             // to errors the validation will fail and the removal will be aborted
        }
    }
}
```

The validation logic is in the `validate()` method. Before calling the entity to be validated, it is injected using `setEntity()`. If messages are added to the errors object the validation will fail and the entity will not be removed. You have to add the error message in the *Invoicing/il8n/Invoicing-messages_en.properties* file, just as in listing 6.13.

Listing 6.13 Internationalization for error in Invoicing-messages_en.properties

```
cannot_delete_order_with_invoice=An order with an invoice cannot be deleted
```

If you try to remove an order with an associated invoice now, you will get an error message and the removal will be rejected.

You can see that using an `@RemoveValidator` is not difficult but verbose. You have to write a full new class to add a simple “if”. Let's examine a briefer alternative.

6.1.7 Validating on removal with a JPA callback method

We're going to try another, maybe simpler, way to do this removal validation just by moving the validation logic from the validator class to the Order entity itself, in this case in a `@PreRemove` method.

First, remove the `OrderRemoveValidator` class from your project. Also remove the `@RemoveValidator` annotation from your Order entity (listing 6.14).

Listing 6.14 Removing the `@RemoveValidator` annotation from the Order entity

```
@RemoveValidator(OrderRemoveValidator.class) // Remove the @RemoveValidator
public class Order extends CommercialDocument {
```

We have just removed the validation. Let's add the functionality again, but now inside the Order class itself. Add the `validateOnRemove()` method in listing 6.15 to your Order class.

Listing 6.15 JPA callback method to validate on removing in the Order entity

```
@PreRemove // Just before removing the entity
private void validateOnRemove() {
    if (invoice != null) { // The validation logic
        throw new IllegalStateException( // Throws a runtime exception
            XavaResources.getString( // To get the text message
                "cannot_delete_order_with_invoice"));
    }
}
```

This validation will be processed before the removal of an order. If it fails an `IllegalStateException` is thrown. You can throw any runtime exception in order to abort the removal. You have done the validation with a single method inside the entity.

6.1.8 What's the best way of validating?

You have learned several ways to do validations in your model classes. Which of them is the best one? All of them are valid options. It depends on your circumstances and personal preferences. If you have a validation that is non-trivial and reusable across your application, then to use `@EntityValidator` and `@RemoveValidator` is a good option. On the other hand, if you want to use your model classes from outside OpenXava and without JPA, then the use of validation in setters is better.

In our example we'll use the `@AssertTrue` for the “delivered to be in invoice” validation and `@PreRemove` for the removal validation, because this is the simplest procedure.

6.2 Creating your own Hibernate Validator annotation

The techniques in the previous section are very useful for many validations. Nevertheless, sometimes you will face some validations that are very generic and you will want to reuse them over and over again. In this case to define your own Hibernate Validator annotation can be a good option. Defining a Hibernate Validator is more verbose but usage and reuse is simple; just adding an annotation to your property or class.

We are going to learn how to create a Hibernate Validator.

6.2.1 Using a Hibernate Validator from your entity

Using a Hibernate Validator is very easy. Just annotate your property, as you see in listing 6.16.

Listing 6.16 Using a Hibernate Validator annotation for our property

```
@ISBN // This annotation indicates this property must be validated as an ISBN
private String isbn;
```

By merely adding @ISBN to your property, it will be validated before the entity is saved into the database. Great! The problem is that @ISBN is not included as a built-in constraint in the Hibernate Validator framework. This is not a big deal. If you want an @ISBN annotation, just create it. Indeed, we are going to create the @ISBN validation annotation in this section.

First of all, let's add a new isbn property to Product. Edit your Product class and add it to the code in listing 6.17.

Listing 6.17 New isbn property in Product

```
@Column(length=10)
private String isbn;

public String getIsbn() {
    return isbn;
}

public void setIsbn(String isbn) {
    this.isbn = isbn;
}
```

Update your database schema, and try out your Product module with the browser. Yes, the isbn property is already there. Now, you can add the validation.

6.2.2 Defining your own ISBN annotation

Let's create the @ISBN annotation. First, create a package in your project called org.openxava.invoicing.annotations. Then follow the instructions in figure

6.2 to create a new annotation called ISBN.

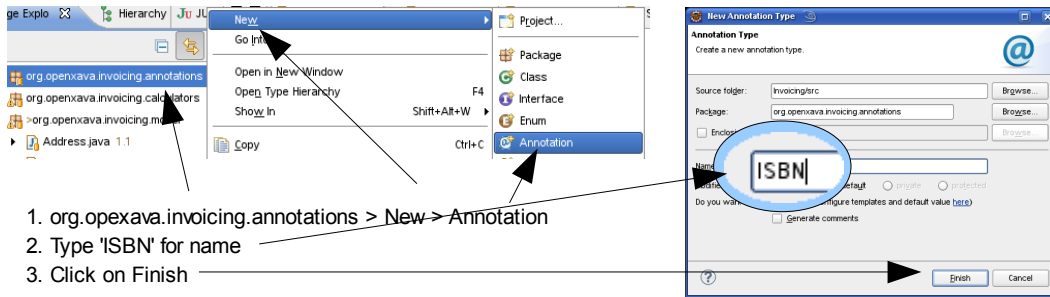


Figure 6.2 Creating the new ISBN annotation with Eclipse

Edit the code of your recently created ISBN annotation and leave it as in listing 6.18.

Listing 6.18 Code for the ISBN annotation

```
package org.opexava.invoicing.annotations; // In 'annotations' package

import java.lang.annotation.*;
import org.hibernate.validator.*;
import org.opexava.invoicing.validators.*;

@ValidatorClass(ISBNValidator.class) // This class contains the validation logic
@Target({ ElementType.FIELD, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface ISBN { // A regular Java annotation definition

    String message() default "ISBN does not exist"; // The message if validation fails

}
```

As you can see, this is a regular annotation definition. The `@ValidatorClass` indicates the class with the validation logic. Let's write the `ISBNValidator` class.

6.2.3 Using Apache Commons Validator to implement the validation logic

We are going to write the `ISBNValidator` class with the validation logic for an ISBN. Instead of writing the ISBN validation logic we'll use the Commons Validator project⁸ from Apache. Commons Validator contains validation algorithms for email addresses, dates, URLs and so on. The *commons-validator.jar* is included by default in OpenXava projects, so you can use it without further configuration.

The code for `ISBNValidator` is in listing 6.19.

Listing 6.19 Initial version of ISBNValidator

⁸ <http://commons.apache.org/validator/>

111 Lesson 6: Advanced validation

```
package org.openxava.invoicing.validators; // In 'validators' package

import org.hibernate.validator.*;
import org.openxava.util.*;
import org.openxava.invoicing.annotations.*;

public class ISBNValidator
    implements Validator<ISBN> { // Must implement Validator<ISBN>

    private static org.apache.commons.validator.ISBNValidator
        validator = // From Commons Validator framework
            new org.apache.commons.validator.ISBNValidator();

    public void initialize(ISBN isbn) {
    }

    public boolean isValid(Object value) { // Contains the validation logic
        if (Is.empty(value)) return true;
        return validator
            .isValid(value.toString()); // Relies on Commons Validator
    }
}
```

As you see, the validator class must implement `Validator` from the `org.hibernate.validator` package. This forces your validator to implement `initialize()` and `isValid()`. The `isValid()` method contains the validation logic. Note that if the value to validate is empty we assume that it is valid. Validating when the value is present is the responsibility of other annotations like `@Required`.

In this case the validation logic is plain vanilla, because we only call the ISBN validator from the Apache Commons Validator project.

`@ISBN` is ready to be used. Just annotate your `isbn` property with it. See it in listing 6.20.

Listing 6.20 The isbn property annotated with `@ISBN`

```
@Column(length=10) @ISBN
private String isbn;
```

Now, you can test your module, and verify that the ISBN values you enter are validated correctly. Congratulations, you have written your first Hibernate Validator. It's not so difficult. One annotation, one class.

This `@ISBN` is good enough for use in real life. Nevertheless, we'll try to improve it, simply to have the chance to experiment with a few interesting possibilities.

6.2.4 Call to a REST web service to validate the ISBN

Though most validators have simple logic, you can create validator with complex logic if necessary. For example, in the case of our ISBN, we want, not only to verify the correct format, but also to check that a book with that ISBN actually exists. A way to do this is by using web services.

As you already know, a web service is a functionality hosted in web servers and can be called by a program. The traditional way to develop and use web services is by means of WS-* standards, like SOAP, UDDI, etc., although, today, a simpler way to develop web services has arisen: REST. The basic idea of REST is to use the already existing “way to work” of the internet for inter-program communication. Calling a REST service consists of using a regular web URL to get a resource from a web server; this resource is usually data in XML, HTML, JSON or any other format. In other words, the programs use the internet just as regular users with their browsers.

There are a lot of sites with SOAP and REST web services that enable us to consult a book ISBN, but usually they are not free. So, we are going to use a cheaper alternative solution, that is, to call a normal web site to do an ISBN search, and to examine the resulting page to determine if the search has succeeded. Something like a pseudo-REST web service.

To call the web page we'll use the HtmlUnit⁹ framework. Though the main goal of this framework is to create tests for your web applications, you can use it to read any web page. We'll use it because it's easier to use than other libraries used for the same purpose, as the Apache Commons HttpClient. See how easy it is to read a web page with HtmlUnit in listing 6.21.

Listing 6.21 Reading a web page using HtmlUnit

```
WebClient client = new WebClient();
HtmlPage page = (HtmlPage) client.getPage("http://www.openxava.org/");
```

After that, you can use the page object to manipulate the read page.

OpenXava uses HtmlUnit as an underlying framework for testing, so it is included with OpenXava. However, by default it's not included in OpenXava applications, so you have to include it yourself in your application. To do so, copy the files *htmlunit.jar*, *commons-httpclient.jar*, *commons-codec.jar*, *htmlunit-core-js.jar*, *commons-lang.jar*, *xercesImpl.jar*, *xalan.jar*, *cssparser.jar*, *sac.jar* and *nekohtml.jar*, from the *OpenXava/lib* folder to *Invoicing/web/WEB-INF/lib* folder. After copying these files refresh the Invoicing project pressing F5.

Let's modify ISBNValidator to use this REST service. See the result in listing

⁹ <http://htmlunit.sourceforge.net/>

6.22.

Listing 6.22 ISBNValidator that uses a REST web service

```

package org.openxava.invoicing.validators;

import org.apache.commons.logging.*;
import org.hibernate.validator.*;
import org.openxava.util.*;
import org.openxava.invoicing.annotations.*;

import com.gargoylesoftware.htmlunit.*; // To use HtmlUnit
import com.gargoylesoftware.htmlunit.html.*; // To use HtmlUnit

public class ISBNValidator implements Validator<ISBN> {

    private static Log log = LogFactory.getLog(ISBNValidator.class);
    private static org.apache.commons.validator.ISBNValidator
        validator =
            new org.apache.commons.validator.ISBNValidator();

    public void initialize(ISBN isbn) {
    }

    public boolean isValid(Object value) {
        if (Is.empty(value)) return true;
        if (!validator.isValid(value.toString())) return false;
        return isbnExists(value); // Here we do the REST call
    }

    private boolean isbnExists(Object isbn) {
        try {
            WebClient client = new WebClient();
            HtmlPage page = (HtmlPage) client.getPage( // We call
                "http://www.bookfinder4u.com/" + // bookdiner4u
                "IsbnSearch.aspx?isbn=" + // using an URL for searching
                isbn + "&mode=direct"); // by ISBN
            return page.asText() // Tests if the resulting page contains the
                .indexOf("ISBN: " + isbn) >= 0; // searched ISBN
        }
        catch (Exception ex) {
            log.warn("Impossible to connect to bookfinder4u" +
                "to validate the ISBN. Validation fails", ex);
            return false; // If there are errors we assume that validation fails
        }
    }
}

```

We simply open the URL with the ISBN as the request parameter. If the resulting page contains the ISBN value, then the search has been successful. Otherwise the search has failed. The `page.asText()` method returns the content of the HTML page without the HTML tags, that is, it only contains the text info.

You can use this trick with any site that allows you to do searches. Thus you can virtually consult millions of web site from inside your application. In a more pure REST web service the result will be an XML document instead of HTML,

but usually you will have to pay some subscription fees.

Try out your application now and you'll see that validation will fail if you enter a non-existent ISBN.

6.2.5 Adding attributes to your annotation

It's a good idea to create a new Hibernate Validator annotation if you reuse the validation several times, usually across several projects. To improve the reusability you may want to parametrize the validation code. For example, for your current project to do the search in www.bookfinder4u.com for ISBN is OK, but in another project, or even in another entity of your current project, you do not want to call this particular URL. The code of the annotation has to be more flexible.

This flexibility can be achieved by attributes. For example, we can add a boolean search attribute to our ISBN annotation in order to switch on or off the internet search for validation. To implement this functionality, just add the search attribute to the ISBN annotation code, like in listing 6.23.

Listing 6.23 ISBN annotation with search attribute

```
public @interface ISBN {

    boolean search() default true; // To (de)activate web search on validate
    String message() default "ISBN does not exist";

}
```

This new search attribute can be read from the validator class. See it in listing 6.24.

Listing 6.24 ISBNValidator with the search attribute

```
public class ISBNValidator implements Validator<ISBN> {

    ...

    private boolean search; // Stores the search option

    public void initialize(ISBN isbn) { // Read the annotation attributes values
        this.search = isbn.search();
    }

    public boolean isValid(Object value) {
        if (Is.empty(value)) return true;
        if (!validator.isValid(value.toString())) return false;
        return search?isbnExists(value):true; // Using search
    }

    ...

}
```

Here you see the use of the `initialize()` method: the source annotation can be used to initialize the validator, in this case simply by storing the `isbn.search()` value to evaluate it in `isValid()`.

Now you can choose whether you want to call our pseudo-REST service or skip the ISBN validation. See listing 6.25.

Listing 6.25 Using @ISBN with search attribute

```
@ISBN(search=false) // In this case no internet search is done to validate the ISBN
private String isbn;
```

Using this simple method you can add any attribute you need to add more flexibility to your ISBN annotation.

Congratulations! You have learned how to create your own Hibernate Validator annotation, and by the way, to use the useful `HtmlUnit` tool.

6.3 JUnit tests

Our goal is not to develop a huge quantity of software, but to create quality software. At the end of the day, if you create quality software you will deliver more functionality, because you will spend more time on new and exciting things and less time on debugging legions of bugs. And you know that the only way to quality is automated testing, so let's update our test code.

6.3.1 Testing validation for adding to a collection

Recall that we have refined the code in a way that the user cannot assign orders to an invoice if the orders are not marked as delivered yet. After that, your current `testAddOrders()` of `InvoiceTest` can fail, because it tries to add the first order, and this first order might not be marked as delivered yet.

Let's modify the test method to run correctly and also to test your new validation functionality. See listing 6.26.

Listing 6.26 testAddOrders() of InvoiceTest now tests validation adding orders

```
public void testAddOrders() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number");
    execute("Mode.detailAndFirst");
    execute("Sections.change", "activeSection=1");
    assertCollectionRowCount("orders", 0);
    execute("Collection.add",
        "viewObject=xava_view_section1_orders");
    execute("AddToCollection.add", "row=0"); // Now we don't select randomly
}
```

```

checkFirstOrderWithDeliveredEquals("Yes"); // Selects one delivered order
checkFirstOrderWithDeliveredEquals("No"); // Selects one not delivered order
execute("AddToCollection.add"); // We try to add both
assertError( // An error, because the not delivered order cannot be added
    "ERROR! 1 element(s) NOT added to Orders of Invoice");
assertMessage( // A confirm message, because the delivered order has been added
    "1 element(s) added to Orders of Invoice");

assertCollectionRowCount("orders", 1);
checkRowCollection("orders", 0);
execute("Collection.removeSelected",
    "viewObject=xava_view_section1_orders");
assertCollectionRowCount("orders", 0);
}

```

We have modified the part for selecting orders to add. Before we selected the first order, no matter if it's delivered or not. Now we select one order delivered and one order not delivered. In this way we test if the delivered one is added and the not delivered one is rejected.

The missing piece here is the way to check the orders. This is the task of the `checkFirstOrderWithDeliveredEquals()` method. Let's see it in listing 6.27.

Listing 6.27 Method to check orders and selecting them by the 'delivered' column

```

private void checkFirstOrderWithDeliveredEquals(String value)
    throws Exception
{
    int c = getListRowCount(); // The total displayed rows in list
    for (int i=0; i<c; i++) {
        if (value.equals(
            getValueInList(i, 10))) // 10 is the 'delivered' column
        {
            checkRow(i);
            return;
        }
    }
    fail("There must be at least one row with delivered=" + value);
}

```

Here you see a good technique to do a loop over the displayed list elements in order to check them, get data or do whatever you want with the list data.

6.3.2 Testing validation assigning a reference and validation on removal

From the Invoice module the user cannot add orders to an invoice if they are not delivered yet, therefore, from the Order module the user cannot assign an invoice to an order if the order is not delivered. That is, we have to test the other side of the association too. We'll do it by modifying the existing `testSetInvoice()` of `OrderTest`.

Moreover, we'll use this case to test the remove validation we introduced in

117 Lesson 6: Advanced validation

sections 6.1.6 and 6.1.7. There we modified the application to prevent the user from removing an order which has an invoice associated with it. Now we will test this restriction.

The revision of `testSetInvoice()` with all these enhancements is printed in listing 6.28.

Listing 6.28 `testSetInvoice()` now tests regular and on remove validation

```
public void testSetInvoice() throws Exception {
    assertListNotEmpty();
    execute("List.orderBy", "property=number"); // To set the list order
    execute("Mode.detailAndFirst");
    assertValue("delivered", "false"); // The order must be not delivered
    execute("Sections.change", "activeSection=1");
    assertValue("invoice.number", "");
    assertValue("invoice.year", "");
    execute("Reference.search",
        "keyProperty=invoice.year");
    execute("List.orderBy", "property=number");
    String year = getValueInList(0, "year");
    String number = getValueInList(0, "number");
    execute("ReferenceSearch.choose", "row=0");
    assertValue("invoice.year", year);
    assertValue("invoice.number", number);

    // Not delivered order cannot have invoice
    execute("CRUD.save");
    assertErrorsCount(1); // We cannot save because it is not delivered
    setValue("delivered", "true");
    execute("CRUD.save"); // With delivered=true we can save the order
    assertNoErrors();

    // Order with invoice cannot be deleted
    execute("Mode.list"); // We go to list and
    execute("Mode.detailAndFirst"); // return to detail to load the saved order
    execute("CRUD.delete"); // We cannot delete because it has an invoice associated
    assertErrorsCount(1);

    // Restoring original values
    setValue("delivered", "false");
    setValue("invoice.year", "");
    execute("CRUD.save");
    assertNoErrors();
}
```

The original test only searched for an invoice, but did not even save it. Now, we added test code at the end which tries to save the order with `delivered=false` and with `delivered=true`, in this way we test the validation. After that, we try to delete the order, that has an invoice. Thus we test the validation on removal too.

6.3.3 Testing the custom Hibernate Validator

The last step is to test the ISBN Hibernate Validator, which uses a REST service to do validation. We simply have to write a test case that tries to assign an

incorrect, a nonexistent and a correct ISBN to a product and checks the results for these cases. To do so let's add a `testISBNValidator()` method to `ProductTest`. See it in listing 6.29.

Listing 6.29 `testISBNValidator()` of `ProductTest` tests our Hibernate Validator

```
public void testISBNValidator() throws Exception {
    // Searching the product1
    execute("CRUD.new");
    setValue("number", Integer.toString(product1.getNumber()));
    execute("CRUD.refresh");
    assertValue("description", "JUNIT Product 1");
    assertValue("isbn", "");

    // With incorrect ISBN format
    setValue("isbn", "1111");
    execute("CRUD.save"); // Fails because of format (apache commons validator)
    assertError("1111 is not a valid value for Isbn of " +
        "Product: ISBN does not exist");

    // ISBN does not exist though it has correct format
    setValue("isbn", "1234367890");
    execute("CRUD.save"); // Fails because it does not exist (REST service)
    assertError("1234367890 is not a valid value for Isbn of " +
        "Product: ISBN does not exist");

    // ISBN exists
    setValue("isbn", "0932633439");
    execute("CRUD.save"); // It does not fail
    assertNoErrors();
}
```

Surely the manual testing you were doing during the development of the `@ISBN` validator was like the one above. Therefore, if you write your JUnit test before the application code¹⁰, you can use it as you proceed. This is more efficient than repeating the test procedures by hand using the browser over and over again.

Note that if you use `@ISBN(search=false)` this test will not work because it checks the result of the REST service. So, you have to use the `@ISBN` annotation of the `isbn` property without the `search` attribute in order to run this test successfully.

Now execute all the tests for your Invoicing application to verify that everything works as expected.

6.4 Summary

In this lesson you have learned several ways to do validation in an OpenXava application. Also, you know how to encapsulate the reusable validation logic in

¹⁰ About Test First advantages: <http://www.extremeprogramming.org/rules/testfirst.html>

annotations with custom Hibernate Validators.

Validation is an important part of the logic of your application, and we encourage you to put it into the model, i. e. into your entities. We demonstrated several examples for this technique in the lesson. Sometimes it is more convenient to put logic outside your model classes. You will learn that in the next lessons.